CS 536 Announcements for Tuesday, February 8, 2022

Programming Assignment 1
- Part 2 files due Tuesday, Feb. 8 by 11:59 pm

Last Time
- regular expressions
- regular expressions \(\rightarrow\) DFAs

Today
- language recognition \(\rightarrow\) tokenizers
- scanner generators
- JLex

Next Time
- CFGs

Recall

\[
\text{scanner} = \text{token to regex} + \text{regex to NFA} + \text{NFA to DFA} + \text{DFA to code}
\]
Regex to DFA

We now can do:

We can add one more step: **optimize DFA**

**Theorem:** For every DFA $M$, there exists a unique equivalent smallest DFA $M^*$ that recognizes the same language as $M$.

**To optimize:**
- remove unreachable states
- remove dead states
- merge equivalent states

**But what's so great about DFAs?**

**Recall:** state-transition function $(\cdot)$ can be expressed as a table

- very efficient array representation

- efficient algorithm for running (any) DFA

```plaintext
s = start state
while (more input) {
    c = read next char
    s = table[s][c]
}
if s is final, accept
else reject
```

What else do we need?
Table-driven DFA → tokenizer

FSMs – only check for language membership of a string

scanner needs to
  • recognize a stream of many different tokens using the longest match
  • know what was matched

Idea: augment states with actions that will be executed when state is reached
Scanner Generator Example

**Language description:**
consider a language consisting of two statements

- assignment statements: \( \text{ID} = \text{expr} \)
- increment statements: \( \text{ID} += \text{expr} \)

where \( \text{expr} \) is of the form:

- \( \text{ID} + \text{ID} \)
- \( \text{ID} ^ \text{ID} \)
- \( \text{ID} < \text{ID} \)
- \( \text{ID} \leq \text{ID} \)

and \( \text{ID} \) are identifiers following C/C++ rules (can contain only letters, digits, and underscores; can't start with a digit)

**Tokens:**

<table>
<thead>
<tr>
<th>Token</th>
<th>Regular expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASSIGN</td>
<td></td>
</tr>
<tr>
<td>INCR</td>
<td></td>
</tr>
<tr>
<td>PLUS</td>
<td></td>
</tr>
<tr>
<td>EXP</td>
<td></td>
</tr>
<tr>
<td>LESSTHAN</td>
<td></td>
</tr>
<tr>
<td>LEQ</td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td></td>
</tr>
</tbody>
</table>

**Combined DFA**
**State-transition table**

<table>
<thead>
<tr>
<th></th>
<th>=</th>
<th>+</th>
<th>^</th>
<th>&lt;</th>
<th>_</th>
<th>letter</th>
<th>digit</th>
<th>EOF</th>
<th>none of these</th>
</tr>
</thead>
<tbody>
<tr>
<td>S₀</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```java
do {
    read char
    perform action / update state
    if (action was to return a token) {
        start again in start state
    }
} while not(EOF or stuck)
```
Lexical analyzer generators
(aka scanner generators)

Formally define transformation from regex to scanner

Tools written to synthesize a lexer automatically
- Lex : UNIX scanner generator, builds scanner in C
- Flex : faster version of Lex
- JLex : Java version of Lex

JLex

Declarative specification

Input: set of regular expressions + associated actions

Output: Java source code for a scanner

Format of JLex specification
3 sections separated by %%
- user code section
- directives
- regular expression rules
JLex example

// This file contains a complete JLex specification for a very small example.

// User Code section: For right now, we will not use it.

%%

DIGIT= [0-9]
LETTER= [a-zA-Z]
WHITESPACE= [\040\t\n]

%state SPECIALINTSTATE

%implements java_cup.runtime.Scanner
%function next_token
%type java_cup.runtime.Symbol

%eofval{
    System.out.println("All done");
    return null;
}%eofval

%line

%

({LETTER}|"_")({DIGIT}|{LETTER}|"_")* {
    System.out.println(yyline+1 + ": ID " + yytext());
}

"=" {
    System.out.println(yyline+1 + ": ASSIGN");
}

"+" {
    System.out.println(yyline+1 + ": PLUS");
}

"^" {
    System.out.println(yyline+1 + ": EXP");
}

"<" {
    System.out.println(yyline+1 + ": LESSTHAN");
}

"+=" {
    System.out.println(yyline+1 + ": INCR");
}

"<=" {
    System.out.println(yyline+1 + ": LEQ");
}

{WHITESPACE}* {
    }

. {
    System.out.println(yyline+1 + ": bad char");
}
Regular expression rules section

Format: `<regex>{code}` where `<regex>` is a regular expression for a single token
- can use macros from Directives section – surround with curly braces `{ }`
- characters represent themselves (except special characters)
- characters inside " " represent themselves (except \"")
- . matches anything

Regular expression operators:   |   *   +   ?   ( )
Character class operators:    -   ^   \ 

Using scanner generated by JLex in a program

```java
// inFile is a FileReader initialized to read from the
// file to be scanned
Yylex scanner = new Yylex(inFile);
try {
    scanner.next_token();
} catch (IOException ex) {
    System.err.println(
        "unexpected IOException thrown by the scanner");
    System.exit(-1);
}
```